Lecture #11: Sequences (II) and Data Abstraction

Announcements

- First test is on Wednesday, 5–7PM (PT).
- LOST section at 4PM will focus extensively on reverse environment diagrams and recursion for midterm review.
- Our indefatigable TAs have created study guides for the upcoming test. See Piazza @808.
- On Saturday (2/13), there are is also an upcoming CSM midterm review from 11AM–12:30PM (Piazza @841), and an HKN review session from 2:30PM–5:30PM (Piazza @827).
- Ask questions on the Piazza thread for today's lecture (Piazza @882).

Multiple Variables

- The iteration variable in a **for** can actually be like the left-hand side of an assignment.
- One can write

```
>>> L = [ (1, 9), (2, 2), (5, 6), (3, 3) ]
>>> same = 0
>>> for x, y in L:
...     if x == y:
...         same += 1
>>> same
2
```

- The elements of L are themselves tuples, so we get the effect of the series of assignments:

```
x, y = (1, 9)
x, y = (2, 2)
...
```


Two Iterations at Once!

- The predefined `zip` function combines multiple sequences:

```
>>> list(zip([1, 2, 5, 3], [9, 2, 6, 3, 10]))
[(1, 9), (2, 2), (5, 6), (3, 3)]
>>> # Length of result is that of shortest sequence
>>> list(zip([1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12, 15]))
[(1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12)]
```

- (**zip** actually returns a *generator*, which we'll cover later.)
- Together with multiple assignments in a **for** loop, gives a way of iterating through sequences in lockstep:

```
>>> beasts = ["aardvark", "axolotl", "gnu", "hartebeest"]
>>> for n, animal in zip(range(1, 5), beasts):
...     print(n, animal)
1 aardvark
2 axolotl
3 gnu
4 hartebeest
```

Modifying Lists

- Lists are mutable sequences. One can assign to elements and to slices.

```
>>> L = [1, 2, 3, 4, 5]
>>> L[2] = 6
>>> L
[1, 2, 6, 4, 5]
>>> L[1:3] = [9, 8]
>>> L
[1, 9, 8, 4, 5]
>>> L[2:4] = []            # Deleting elements
>>> L
[1, 9, 5]
>>> L[1:1] = [2, 3, 4, 5]  # Inserting elements
>>> L
[1, 2, 3, 4, 5, 9, 5]
>>> L[len(L):] = [10, 11]  # Appending
>>> L
[1, 2, 3, 4, 5, 9, 5, 10, 11]
>>> L[0:0] = range(-3, 0)  # Prepending
>>> L
[-3, -2, -1, 1, 2, 3, 4, 5, 9, 5, 10, 11]
```


List Comprehensions

- Full form:

```
[ <expression> for <var> in <sequence expression>
               if <boolean expression> ]
```

where the **if** is optional.

- Example: Squares of the prime numbers up to 100.

```
[ x*x for x in range(101) if isprime(x) ]
```

- The comprehension's value is computed in a new local frame, so that (unlike the **for** statement), the value of `x` is undefined afterwards.
- Actually, one can have multiple **for** clauses, giving the effect of a loop within a loop:

```
>>> [ (a, b) for a in range(10, 13) for b in range(2) ]
[(10, 0), (10, 1), (11, 0), (11, 1), (12, 0), (12, 1)]
```


Exercise I

- Give a one-line expression that takes two lists and returns the number of indices in which both have the same value.

```
def matches(a, b):
    """Return the number of values k such that A[k] == B[k].
    >>> matches([1, 2, 3, 4, 5], [3, 2, 3, 0, 5])
    3
    >>> matches("abdomens", "indolence")
    4
    >>> matches("abcd", "dcba")
    0
    >>> matches("abcde", "edcba")
    1
    """
```


Exercise II

- Fill in the blank to make the comment true.

```
def triangle(n):
    """Assuming N >= 0, return the list consisting of N lists:
    [1], [1, 2], [1, 2, 3], ... [1, 2, ... N].
    >>> triangle(0)
    []
    >>> triangle(1)
    [[1]]
    >>> triangle(5)
    [[1], [1, 2], [1, 2, 3], [1, 2, 3, 4], [1, 2, 3, 4, 5]]
    """
```

An aside: Sequences in Unix

- Many Unix utilities operate on *streams of characters,* which are sequences.
- With the help of pipes, one can do amazing things. One of my favorites:

```
tr -c -s '[:alpha:]' '[\n*]' < FILE | \
sort | \
uniq -c | \
sort -n -r -k 1,1 | \
sed 20q
```

  which prints the 20 most frequently occurring words in *FILE,* with their frequencies, most frequent first.
- The commands actually mean:
  1. Translate non-alphabetic characters to newlines and collapse adjacent newlines.
  2. Sort the result of 1.
  3. Collapse adjacent duplicate lines from 2; prepend duplicate count.
  4. Sort the result of 3 numerically by the duplicate count.
  5. Print the first 20 lines of 4 and then quit.

Data Abstraction

Philosophy

- In the old days, one described programs as hierarchies of actions: *procedural decomposition*.
- Starting in the 1970's, emphasis moved to the data that the functions operate on.
- An *abstract data type (ADT)* (like the pair abstraction from the last lecture) represents some kind of thing and the operations on it.
- We can usefully organize our programs around the ADTs in them.
- For each type, we define an *interface* that describes for users ("clients") of that type of data what operations are available (*API* for *Application Programmer's Interface*).
- Typically, the interface consists of functions.
- The collection of specifications (syntactic and semantic) of these functions constitutes a *specification of the type.*
- We call ADTs *abstract* because clients ideally need not know internals.

Classification

- As an example, last time we defined a *pair* type as a set of functions.
- Some of these functions fall into common categories:
  - *Constructors* create new items of the type: e.g., `pair`.
  - *Accessors* return properties of items of the type: e.g., `left` and `right`.
  - *Mutators* modify items of the type: e.g, `set_left`, and `set_right`.

Rational Numbers

- The book uses "rational number" as an example of an ADT:

```
def make_rat(n, d):         # Constructor
    """The rational number N/D, assuming N, D are integers, D!=0"""

def numer(r):               # Accessor
    """The numerator of rational number R."""

def denom(r):               # Accessor
    """The denominator of rational number R."""
```

- The last two definitions pretend that `r` really is a rational number.
- But from this point of view, the definitions of `numer` and `denom` are problematic. Why?

A Better Specification

- Problem is that "the numerator (denominator) of r" is not well-defined for a rational number.
- If `make_rat` really produced rational numbers, then `make_rat(2, 4)` and `make_rat(1, 2)` ought to be identical. So should `make_rat(1, -1)` and `make_rat(-1, 1)`.
- So a better specification would be

```
def numer(r):
    """The numerator of rational number R in lowest terms."""

def denom(r):
    """The denominator of rational number R in lowest terms.
       Always positive."""
```


Additional Operations

- Rationals, being numbers, should support numeric and other operations:

```
def add_rat(x, y):
    """The sum of rational numbers X and Y."""

def mul_rat(x, y):
    """The product of rational numbers X and Y."""

def str_rat(r):
    """Return R as a string containing a rational fraction.
    >>> str_rat(make_rat(2, 4))
    1/2
    >>> str_rat(make_rat(3, 1))
    3
    """

def equal_rat(x, y):
    """Return True iff X and Y are equal rational numbers."""
```


Using Rationals

- For example, we can now write functions that (aside from syntax) manipulate rationals as we would other kinds of number:

```
def approx_harmonic_number(n):
    """Return an approximation to 1 + 1/2 + 1/3 + ... + 1/N."""
    s = 0.0
    for k in range(1, n + 1):
        s = s + 1 / k
    return s

def exact_harmonic_number(n):
    """Return 1 + 1/2 + 1/3 + ... + 1/N as a rational number."""
    s = make_rat(0, 1)
    for k in range(1, n + 1):
        s = add_rat(s, make_rat(1, k))
    return s
```

- Later, we'll see how to make the syntax (nearly) identical as well.

Representing Rationals

- Either the `pair` abstraction from last time (represented by functions) or Python tuples or lists can represent rational numbers.

```
from math import gcd

def make_rat(n, d):
    """The rational number N/D, assuming N, D are integers, D!=0"""
    g = gcd(n, d)
    n //= g; d //= g
    return (n, d)

def numer(r):
    """The numerator of rational number R in lowest terms."""
    return r[0]

def denom(r):
    """The denominator of rational number R in lowest terms.
       Always positive."""
    return r[1]
```


Implementing Additional Operations

- One possibility for `add_rat`:

```
from math import gcd

def make_rat(n, d):
    """The rational number N/D, assuming N, D are integers, D!=0"""
    g = gcd(n, d)
    n //= g; d //= g
    return (n, d)
...
def add_rat(x, y):
    n0, d0 = x
    n1, d1 = y
    n = n0 * d1 + n1 * d0
    d = d0 * d1
    g = gcd(n, d)
    n //= g; d //= g
    return (n, d)
```

- Comments?

Abstraction Violations and DRY

- Having created an abstraction (`make_rat`, `numer`, `denom`), use it:
  1. Later changes of representation will affect less code.
  2. Code will be clearer, since well-chosen names in the API make intent clear.
- Better implementations of the additional operations might then be

```
def add_rat(x, y):
    return make_rat(numer(x) * denom(y) + numer(y) * denom(x),
                    denom(x) * denom(y))

def mul_rat(x, y):
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))

def str_rat(r):  # (For fun: a little new Python string magic)
    return str(numer(r)) if denom(r) == 1 else f"{numer(r)}/{denom(r)}"

def equal_rat(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```


Layers of Abstraction

- So we can divide up our operations like this:

Primitives	`(...,...)`	`...[...]`		
Representation	`make_rat`	`numer`	`denom`	
Derived Operations:	`add_rat`	`mul_rat`	`print_rat`	`equal_rat`
User Program:	`exact_harmonic_number`			

- The dark lines here represent *abstraction barriers*.
- Layers above a barrier use nothing beneath them
- Layers below a barrier use only the operations from the layer above, which we say are *exported* to it.

Abstraction Violations

- So as well as violating DRY, the implementation

```
def add_rat(x, y):
    n0, d0 = x     # NAUGHTY
    n1, d1 = y     # NAUGHTY
    n = n0 * d1 + n1 * d0
    d = d0 * d1
    g = gcd(n, d)
    n //= g; d //= g
    return (n, d)  # NAUGHTY
```

  violates the abstraction barrier above `add_rat`, reaching into the details of the representation instead of relying on its exported operations.